

A Fast Algorithm for Permutation Pattern Matching Based on Alternating Runs*

Marie-Louise Bruner and Martin Lackner

marie-louise.bruner@tuwien.ac.at
lackner@dbai.tuwien.ac.at

Vienna University of Technology

The NP-complete PERMUTATION PATTERN MATCHING problem asks whether a permutation P can be matched into a permutation T . A matching is an order-preserving embedding of P into T . We present a fixed-parameter algorithm solving this problem with an exponential worst-case runtime of $\mathcal{O}^*(1.79^{\text{run}(T)})$, where $\text{run}(T)$ denotes the number of alternating runs of T . This is the first algorithm that improves upon the $\mathcal{O}^*(2^n)$ runtime required by brute-force search without imposing restrictions on P and T . Furthermore we prove that – under standard complexity theoretic assumptions – such a fixed-parameter tractability result is not possible for $\text{run}(P)$.

1. Introduction

The concept of pattern avoidance (and, closely related, pattern matching) in permutations arose in the late 1960ies. It was in an exercise of his *Fundamental algorithms* [15] that Knuth asked which permutations could be sorted using a single stack. The answer is simple: These are exactly the permutations avoiding the pattern 231 and they are counted by the Catalan numbers. By avoiding (resp. containing) a certain pattern the following is meant: The permutation $\pi = 53142$ (written in one-line representation) contains the pattern 231, since the subsequence 342 of π is order-isomorphic to 231. We call the subsequence 342 a matching of 231 into π . On the other hand, π avoids the pattern 123 since it contains no increasing subsequence of length three. Since 1985, when the first systematic study of *Restricted Permutations* [21] was published by Simion and

*A short version of this paper is going to appear in the proceedings of the 13th Scandinavian Symposium and Workshops on Algorithm Theory, SWAT 2012.

Schmidt, the area of pattern avoidance in permutations has become a rapidly growing field of discrete mathematics, more specifically of (enumerative) combinatorics [5, 14].

This paper takes the viewpoint of computational complexity. Computational aspects of pattern avoidance, in particular the analysis of the PERMUTATION PATTERN MATCHING (PPM) problem, have received far less attention than enumerative questions until now. The PPM problem is defined as follows:

PERMUTATION PATTERN MATCHING (PPM)

Instance: A permutation T (the text) of length n and a permutation P (the pattern) of length $k \leq n$.

Question: Is there a matching of P into T ?

In [6] it was shown that PPM is in general NP-complete. From this result follows a trivial brute-force algorithm checking every length k subsequence of T . Its runtime is in $\mathcal{O}^*(2^n)$, i.e. is bounded by $2^n \cdot \text{poly}(n)$. To the best of our knowledge, no algorithm with a runtime of $\mathcal{O}^*((2 - \epsilon)^n)$ without restrictions on P and T is known yet. If such restrictions are imposed, improvements have been achieved. There are polynomial time algorithms in case of a *separable* pattern [2, 6, 13, 19]. Separable permutations avoid both 3142 and 2413. In case P is the identity $12 \dots k$, PPM consists of looking for an increasing subsequence of length k in the text – this is a special case of the LONGEST INCREASING SUBSEQUENCE problem. This problem can be solved in $\mathcal{O}(n \log n)$ -time for sequences in general [20] and in $\mathcal{O}(n \log \log n)$ -time for permutations [9, 17]. PPM can be solved in $\mathcal{O}(n \log n)$ -time for all patterns of length four [2]. An $\mathcal{O}(k^2 n^6)$ -time algorithm is presented in [12] for the case that both the text and the pattern are 321-avoiding.

In this paper we tackle the problem of solving PPM faster than $\mathcal{O}^*(2^n)$ for arbitrary P and T . We achieve this by exploiting the decomposition of permutations into alternating runs. As an example, the permutation $\pi = 53142$ has three alternating runs: 531 (down), 4 (up) and 2 (down). We denote this number of ups and downs in a permutation π by $\text{run}(\pi)$. Alternating runs are a fundamental permutation statistic and had been studied already in the late 19th century by André [4]. An important result was the characterization of the distribution of $\text{run}(\pi)$ in a random permutation: asymptotically, $\text{run}(\pi)$ is normal with mean $\frac{1}{3}(2|\pi| - 1)$ [16]. Despite the importance of alternating runs within the study of permutations, the connection to PPM has so far not been explored.

Contributions. In detail the contributions of this paper are the following:

- We present a fixed-parameter algorithm for PPM with an exponential runtime of $\mathcal{O}^*(1.79^{\text{run}(T)})$. Since the combinatorial explosion is confined to $\text{run}(T)$, this algorithm performs especially well when T has few alternating runs. Indeed, the runtime depends only polynomially on n , the length of T .
- Since $\text{run}(T) \leq n$, this algorithm also solves PPM in time $\mathcal{O}^*(1.79^n)$. This is a major improvement over the brute-force algorithm.

- Furthermore, we analyze this algorithm with respect to $\text{run}(P)$. We obtain a runtime of $\mathcal{O}^*\left((n^2/2\text{run}(P))^{\text{run}(P)}\right)$. In the framework of parameterized complexity theory this runtime proves XP membership for $\text{run}(P)$.
- We also prove that an algorithm presented in [1] has a runtime of $\mathcal{O}(n^{1+\text{run}(P)})$. This is achieved by proving an inequation that bounds the pathwidth of a certain graph generated by a permutation by the number of alternating runs of this permutations.
- Finally, we prove that these XP results cannot be substantially improved. We prove that – under standard complexity theoretic assumptions – no fixed-parameter algorithm exists with respect to $\text{run}(P)$, i.e. no algorithm with runtime $\mathcal{O}^*(c^{\text{run}(P)})$ for some constant c may be hoped for.

Related work. There is further related work to be mentioned. In [1] an algorithm for PPM with a runtime of $\mathcal{O}(n^{0.47k+o(k)})$ is presented.

The LONGEST COMMON PATTERN problem is to find a longest common pattern between two permutations T_1 and T_2 , i.e. a pattern P of maximal length that can be matched both into T_1 and T_2 . This problem is a generalization of PPM since determining whether the longest common pattern between T_1 and T_2 is T_1 is equivalent to PPM. In [7] a polynomial time algorithm for the LONGEST COMMON PATTERN problem is presented for the case that one of the two permutations T_1 and T_2 is separable. A generalization of this problem, the so called LONGEST COMMON \mathcal{C} -PATTERN problem was introduced in [8]. This problem consists of finding the longest common pattern among several permutations belonging to a class \mathcal{C} of permutations. For the case that \mathcal{C} is the class of all separable permutations and that the number of input permutations is fixed, the problem was shown to be polynomial time solvable.

For a class of permutations X the LONGEST X -SUBSEQUENCE (LXS) problem is to identify in a given permutation T its longest subsequence that is isomorphic to a permutation of X . In [3] polynomial time algorithms for many classes X are described, in general however LXS is NP-hard.

Organization of the paper. Section 2 contains essential definitions for permutations and parameterized complexity theory. The main section, Section 3, describes the algorithm and proves runtime bounds and correctness. The hardness results can be found in Section 4. We conclude with possible future research directions in Section 5 and acknowledgments in Section 6.

2. Preliminaries

2.1. Permutations

For any $m \in \mathbb{N}$, let $[m]$ denote the set $\{1, \dots, m\}$ and $[0, m]$ denote $\{0, 1, \dots, m\}$. A permutation π on the set $[m]$ can be seen as the sequence $\pi(1), \pi(2), \dots, \pi(m)$. Viewing

permutations as sequences allows us to speak of *subsequences* of a permutation. We speak of a *contiguous subsequence* of π if the sequence consists of contiguous elements in π .

Definition 2.1. Let P (the pattern) be a permutation of length k . We say that the permutation T (the text) of length n contains P as a pattern or that P can be matched into T if we can find a subsequence of T that is order-isomorphic to P . If there is no such subsequence we say that T avoids the pattern P . Matching P into T thus consists in finding a monotonically increasing map $\varphi : [k] \rightarrow [n]$ so that the sequence $\varphi(P)$, defined as $(\varphi(P(i)))_{i \in [k]}$, is a subsequence of T .

Every permutation π on $[m]$ defines a total order \prec_π on $[m]$. We write $i \prec_\pi j$ iff $\pi^{-1}(i) < \pi^{-1}(j)$, i.e. the value i stands to the left of the value j in π . When considering the minimum (maximum) of a subset $\mathcal{S} \subseteq [m]$ with respect to \prec_π , we write $\min_\pi \mathcal{S}$ ($\max_\pi \mathcal{S}$).

We discern two types of local extrema in permutations: valleys and peaks. A *valley* of a permutation π is an element $\pi(i)$ for which it holds that $\pi(i-1) > \pi(i)$ and $\pi(i) < \pi(i+1)$. If $\pi(i-1)$ or $\pi(i+1)$ is not defined, we still speak of valleys. The set $\text{Val}(\pi)$ contains all valleys of π . Similarly, a *peak* denotes an element $\pi(i)$ for which it holds that $\pi(i-1) < \pi(i)$ and $\pi(i) > \pi(i+1)$.

Valleys and peaks partition a permutation into contiguous monotone subsequences, so-called (*alternating*) *runs*. The first run of a given permutation starts with its first element (which is also the first local extremum) and ends with the second local extremum. The second run starts with the following element and ends with the third local extremum. Continuing in this way, every element of the permutation belongs to exactly one alternating run. Observe that every alternating run is either increasing or decreasing. We therefore distinguish between *runs up* and *runs down*. Note that runs up always end with peaks and runs down always end with valleys. The parameter $\text{run}(\pi)$ counts the number of alternating runs in π . Hence $\text{run}(\pi) + 1$ equals the number of local extrema in π . These definitions can be analogously extended to subsequences of permutations.

Example 2.2. In the permutation 1 8 12 4 7 11 6 3 2 9 5 10 the valleys are 1, 4, 2 and 5 and the peaks are 12, 11, 9 and 10. A decomposition into alternating runs is given by: 1 8 12|4|7 11|6 3 2|9|5|10. A graphical representation can be found in Figure 1 on page 7. −

2.2. Parameterized complexity theory

We give the relevant definitions of parameterized complexity theory. In contrast to classical complexity theory, a parameterized complexity analysis studies the runtime of an algorithm with respect to an additional parameter and not just the input size $|I|$. Therefore every parameterized problem is considered as a subset of $\Sigma^* \times \mathbb{N}$, where Σ is the input alphabet. An instance of a parameterized problem consequently consists of an input string together with a positive integer p , the parameter.

Definition 2.3. A parameterized problem P is fixed-parameter tractable (or in FPT) if there is a computable function f and an integer c such that there is an algorithm solving P in time $\mathcal{O}(f(p) \cdot |I|^c)$.

The algorithm itself is also called fixed-parameter tractable. In this paper we want to focus on the exponential runtime of algorithms, i.e. the function f , and therefore use the \mathcal{O}^* notation which neglects polynomial factors.

A central concept in parameterized complexity theory are *fixed-parameter tractable reductions*.

Definition 2.4. Let $L_1, L_2 \subseteq \Sigma^* \times \mathbb{N}$ be two parameterized problems. An fpt-reduction from L_1 to L_2 is a mapping $R : \Sigma^* \times \mathbb{N} \rightarrow \Sigma^* \times \mathbb{N}$ such that

- $(I, k) \in L_1$ iff $R(I, k) \in L_2$.
- R is computable by an fpt-algorithm.
- There is a computable function g such that for $R(I, k) = (I', k')$, $k' \leq g(k)$ holds.

FPT is the parameterized equivalent of PTIME. Other important complexity classes in the framework of parameterized complexity are $W[1] \subseteq W[2] \subseteq \dots$, the W-hierarchy. For our purpose, only the class $W[1]$ is relevant. It is conjectured (and widely believed) that $W[1] \neq \text{FPT}$. Therefore showing $W[1]$ -hardness can be considered as evidence that the problem is not fixed-parameter tractable.

Definition 2.5. The class $W[1]$ is defined as the class of all problems that are fpt-reducible to the following problem.

TURING MACHINE ACCEPTANCE

Instance: A nondeterministic Turing machine with its transition table, an input word x and a positive integer k .
Parameter: k
Question: Does the Turing machine accept the input x in at most k steps?

Definition 2.6. A parameterized problem is in XP if it can be solved in time $\mathcal{O}(|I|^{f(k)})$ where f is a computable function.

All the aforementioned classes are closed under fpt-reductions. The following relations between these complexity classes are known:

$$\text{FPT} \subseteq W[1] \subseteq W[2] \subseteq \dots \subseteq \text{XP}$$

Further details can be found e.g. in the monographs [10, 11, 18].

3. The alternating run algorithm

We start with an outline of the alternating run algorithm. Its description consists of two parts. In Part 1 we introduce so-called *matching functions*. These functions map runs in P to sequences of adjacent runs in T . The intention behind matching functions is to restrict the search space to certain length k subsequences, namely to those where all elements in a run in P are mapped to elements in the corresponding sequences of runs in T . In Part 2 a dynamic programming algorithm is described. It checks for every matching function whether it is possible to find a compatible matching. This is done by finding a small set of representative elements to which the element 1 can be mapped to, then – for a given choice for 1 – finding representative values for 2, and so on.

Theorem 3.1. *The alternating run algorithm solves PPM in time $\mathcal{O}^*(1.79^{\text{run}(T)})$. Therefore PPM parameterized by $\text{run}(T)$ is in FPT.*

Corollary 3.2. *The alternating run algorithm solves PPM in time $\mathcal{O}^*(1.79^n)$ where n is the length of the text T .*

Throughout this section we will use the input instance (T_{ex}, P_{ex}) which is given by $T_{ex} = 1\ 8\ 12\ 4\ 7\ 11\ 6\ 3\ 2\ 9\ 5\ 10$ and $P_{ex} = 2\ 3\ 1\ 4$ as a running example.

Part 1: Matching functions. We introduce the concept of matching functions. These are functions from $[\text{run}(P)]$, i.e. runs in P , to sequences of adjacent runs in T . For a given matching function F the search space in T is restricted to matchings where an element i contained in the j -th run in P is matched to an element in $F(j)$. Two adjacent runs in P are mapped to sequences of runs that overlap with exactly one run. This overlap is necessary since elements in different runs in P may be matched to elements in the same run in T . More precisely, valleys and peaks in P might be matched to the same run in T as their successors (see the following example).

Example 3.3. In Figure 1 P_{ex} (left-hand side) and T_{ex} (right-hand side) are depicted together with a matching function F . A matching compatible with F is given by 4 6 2 9. We can see that the elements 6 and 2 lie in the same run in T_{ex} even though 3 (a peak) and 1 (its successor) lie in different runs in P_{ex} . \dashv

Definition 3.4. *A matching function F maps an element of $[\text{run}(P)]$ to a subsequence of T . It has to satisfy the following properties for all $i \in [\text{run}(P)]$.*

- (P1) $F(i)$ is a contiguous subsequence of T .
- (P2) If the i -th run in P is a run up (down), $F(i)$ starts with an element following a valley (peak) or the first element in T and ends with a valley (peak) or the last element in T .
- (P3) $F(1)$ starts with the first and $F(\text{run}(P))$ ends with the last element in T .
- (P4) $F(i)$ and $F(i + 1)$ have one run in common: $F(i + 1)$ starts with the leftmost element in the last run in $F(i)$.

possibilities of picking $\text{run}(P) - 1$ (for the first run in P no choice has to be made) runs among the at most $\lceil \text{run}(T)/2 \rceil$ runs up in T . Hence there are at most

$$\binom{\lceil \text{run}(T)/2 \rceil}{\text{run}(P) - 1} \leq 2^{\lceil \text{run}(T)/2 \rceil - 1} < \sqrt{2}^{\text{run}(T)}$$

such functions. The first inequality holds since $\binom{n}{k} < 2^{n-1}$ for all $n, k \in \mathbb{N}$ as can easily be proven by induction over n . \square

Part 2: Finding a matching. When checking whether T contains P as a pattern, it is sufficient to test for all matching functions whether there exists a *compatible* matching. A matching is compatible with a matching function F if an element i contained in the j -th run in P is matched to an element in $F(j)$. This is checked by a dynamic programming algorithm. The algorithm computes the data structure X_κ for each $\kappa \in [k]$. X_κ is a subset of $[0, n]^{\text{run}(P)}$ and contains representative choices for the matching of the largest element in each run in P that is $\leq \kappa$. (X_κ does of course depend on F but we omit this in the notation.)

Let us explain what is meant by representative choices. We search for a compatible matching of P into T by successively determining possible elements for $1, 2, \dots, k$. Given a choice for $\kappa \in [k]$, possible choices for $\kappa + 1$ are necessarily larger. In addition, it is always preferable to choose elements that are as small as possible. To be more precise: if $\nu \in [n]$ has been chosen for $\kappa \in [k]$, we merely need to consider the valleys of the subsequence of T containing all elements larger than ν . Indeed, if any matching of P into T can be found, it is also possible to find a matching that only involves valleys in the above-mentioned subsequences. Therefore our algorithm will only consider such valleys – we call these elements representative. As an example, consider again Figure 1. Here 4 6 3 10 is a matching of P_{ex} into T_{ex} where the elements 3 and 10 are not representative. This can be seen since 3 is not a valley and 10 is not a valley in the subsequence consisting of elements larger than 6. However, this matching can be represented by the matching 4 6 2 9 that only involves representative elements (3 is represented by 2; 10 by 9).

Furthermore, observe that when possible elements for $1, 2, \dots, k$ are successively determined, we move from left to right in runs up and from right to left in runs down. Hence the chosen elements do not only have to be larger than the previously chosen element but also have to lie on the correct side of the previously chosen element in the same run. These observations are captured in the following definition.

Definition 3.6. For a permutation π on $[n]$ and integers $i, j \leq n$, we define $\pi_{U(i,j)}$ ($\pi_{D(i,j)}$) as the subsequence of π consisting of all elements that are right (left) of j and larger than i . Then $URep(\pi, i, j) := \text{Val}(\pi_{U(i,j)})$ (resp. $DRep(\pi, i, j) := \text{Val}(\pi_{D(i,j)})$) corresponds to the set of representative elements for the case of a run up (resp. down).

For an example, see Figure 3 where representative elements are shown for the permutation T_{ex} , $i = 3$ and $j = 2$.

We now describe how the algorithm checks whether there is a matching from P into T compatible with the matching function F . The data structure X_κ consists of $\text{run}(P)$ -tuples with entries in $[0, n]$. The i -th component of a tuple in X_κ is a representative

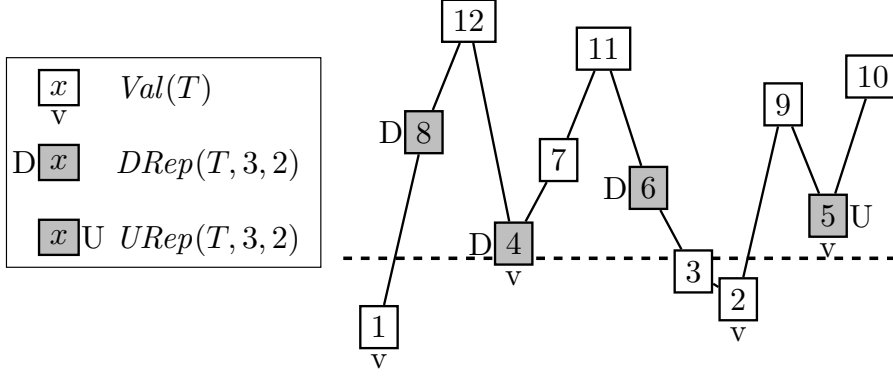


Figure 3: Illustrating Definition 3.6

choice for the largest element $\leq \kappa$ that lies in the i -th run. Limiting X_κ to representative elements allows to bound its size by $1.262^{\text{run}(T)}$. In order to achieve this upper bound, we impose the following conditions (C1) and (C2) and remove unnecessary elements by applying the rules (R1) and (R2). In order to state these conditions and rules, we write $r(\kappa) = i$ iff κ is contained in the i -th run in P . For notational convenience we define $r(0) := 1$.

First, we set $X_0 := \{(0, 0, \dots, 0)\}$. The set X_κ is then constructed from $X_{\kappa-1}$ as follows. Let $\mathbf{x} = (x_1, \dots, x_{\text{run}(P)}) \in X_{\kappa-1}$. We now define $N_{\kappa, \mathbf{x}}$ to be the set of all $\nu \in [n]$ that satisfy (C1) and (C2). This set contains representative elements to which κ may be mapped to for the given \mathbf{x} .

(C1) It has to hold that $\nu \in URep(F(r(\kappa)), x_{r(\kappa-1)}, x_{r(\kappa)})$ in case κ lies in a run up and analogously $\nu \in DRep(F(r(\kappa)), x_{r(\kappa-1)}, x_{r(\kappa)})$ in case κ lies in a run down.

This condition ensures that ν is larger than the previously chosen element for $\kappa - 1$, i.e. larger than $x_{r(\kappa-1)}$. Furthermore, it enforces ν to lie on the correct side of $x_{r(\kappa)}$, the previously chosen element in this run. Instead of considering all such elements in $F(r(\kappa))$ we only take into account representative elements.

(C2) If κ is *not* the largest element in its run in P , there has to exist $\xi \in F(r(\kappa))$ with $\nu < \xi$ and $\nu \prec_T \xi$ for κ appearing in a run up ($\xi \prec_T \nu$ for κ appearing in a run down).

This condition excludes a choice for κ that cannot lead to a matching. A non-maximal element in a run up (down) in P has to be mapped to an element having larger elements to its right (left). We therefore exclude elements in the rightmost (leftmost) run of $F(r(\kappa))$ if this is a run down (up). Condition (C2) is necessary to obtain the runtime bounds for the dynamic programming algorithm.

As an intermediate step let $X'_\kappa := \{\mathbf{x}(\nu) \mid \mathbf{x} \in X_{\kappa-1} \text{ and } \nu \in N_{\kappa, \mathbf{x}}\}$, where $\mathbf{x}(\nu) := (x_1, \dots, x_{r(\kappa)-1}, \nu, x_{r(\kappa)+1}, \dots, x_{\text{run}(P)})$. The tuple $\mathbf{x}(\nu)$ thus differs from \mathbf{x} only at the $r(\kappa)$ -th position. Note that two different elements \mathbf{x} and \mathbf{x}' in $X_{\kappa-1}$ may lead to the

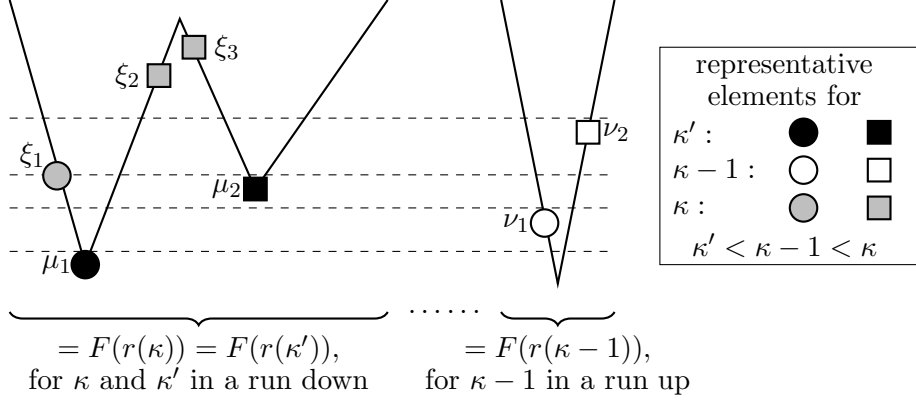


Figure 4: Illustrating Rule (R1)

same element $\mathbf{x}(\nu) = \mathbf{x}'(\nu)$ in X_κ if they only differ in $x_{r(\kappa)}$. Rule (R1) describes how to compute X_κ from X'_κ . Stating it requires the following definition.

Definition 3.7. Let π be a permutation of length n . A subsequence of π consisting of a consecutive run down and run up (formed like a V) is called a vale. If π starts with a run up, this run is also considered as a vale and analogously if π ends with a run down. For two elements $\nu_1, \nu_2 \in [n]$, $\nu_1 \sim \nu_2$ if both lie in the same vale¹. For two k -tuples $\mathbf{x}, \mathbf{y} \in [n]^k$, $\mathbf{x} \sim \mathbf{y}$ if for every $i \in [k]$ it holds that $x_i \sim y_i$. For a fixed set of k -tuples S and $\mathbf{x} \in S$, the equivalence class $[\mathbf{x}]_\sim$ is defined as all $\mathbf{y} \in S$ with $\mathbf{x} \sim \mathbf{y}$.

(R1) We set $X_\kappa := \{\min_{(r(\kappa))}([\mathbf{x}]_\sim) \mid \mathbf{x} \in X'_\kappa\}$, where $\min_{(r(\kappa))}([\mathbf{x}]_\sim)$ picks the tuple in $[\mathbf{x}]_\sim$ with the smallest value at the $r(\kappa)$ -th position. If this minimum is not unique, it arbitrarily picks one candidate.

This rule is the key to prove the $1.2611^{\text{run}(T)}$ upper bound on $|X_\kappa|$. It is based on the observation that it is enough to keep a single tuple for each $[\mathbf{x}]_\sim$. This means that for a set of tuples with coinciding vales it is enough to consider one of them. We provide an intuition about the rule and its correctness in the following example.

Example 3.8. Consider the text permutation schematically represented in Figure 4. We are searching for representative choices for κ , an element lying in a run down. For κ' , the previous element lying in the same run as κ , two representative elements are μ_1 (circle) and μ_2 (square). They lead to one representative element for $\kappa - 1$ each: if μ_1 has been chosen ν_1 is a representative element (circle) and if μ_2 has been chosen ν_2 is one. Following condition (C1), we find three representative elements for κ in $F(r(\kappa))$: ξ_1 (if ν_1 has been chosen), ξ_2 and ξ_3 (if ν_2 has been chosen).

We can now observe that it is not necessary to store all three representative elements for κ . Indeed, in the vale containing ξ_1 and ξ_2 we only need to keep track of ξ_1 since

¹Note that every element in a permutation is contained in exactly one vale.

this is always a better choice than ξ_2 . This can be seen in the following way: In general, elements that lie further to the right (left) in a run down (up) might be preferable since they leave more possibilities for future elements that are to be matched. Within a vale however, the horizontal position does not make any difference, it is only the vertical position that matters. Here, the elements left of ξ_2 and right of ξ_1 are not available for following choices even if we choose ξ_2 since they are smaller than ξ_2 . However, the elements left of ξ_1 that are smaller than ξ_2 are only available if we choose ξ_1 . \dashv

In the case that κ is the largest element in its run, it is enough to consider a single representative element in $F(r(\kappa))$. This is because the position of the element ν is no longer relevant since no further elements have to be chosen in this run. Hence the following data reduction is performed on X_κ .

(R2) Let $M_{\kappa, \mathbf{x}} := \{y_{r(\kappa)} \mid \mathbf{y} \in X_\kappa \wedge (y_i = x_i \ \forall i \neq r(\kappa))\}$. If κ is the largest element in its run in P , each $\mathbf{x} = (x_1, \dots, x_{r(\kappa)-1}, x_{r(\kappa)}, x_{r(\kappa)+1}, \dots, x_{\text{run}(P)}) \in X_\kappa$ is replaced by the tuple $(x_1, \dots, x_{r(\kappa)-1}, \min(M_{\kappa, \mathbf{x}}), x_{r(\kappa)+1}, \dots, x_{\text{run}(P)})$.

As a consequence there are no two tuples in X_κ that only differ at the $r(\kappa)$ -th position in this case.

Termination. For a given matching function F , the algorithm described in Part 2 terminates as soon as we have reached X_k . Observe that X_k is always empty if a previous X_κ was empty. If for any F the data structure X_k is non-empty, P can be matched into T .

Example 3.9. Let us demonstrate with the help of a simple example how the alternating run algorithm works. Consider the text T_{ex} and the pattern P_{ex} . In this example we consider the matching function F represented in Figure 1. Figure 5 depicts a successful run of the algorithm finding the matching 4629. \dashv

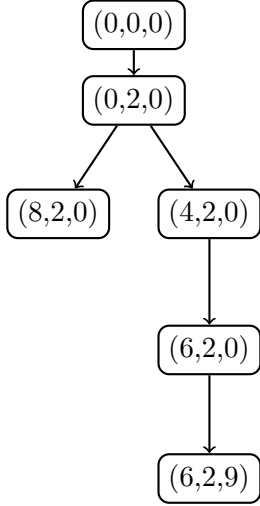
3.1. Correctness

Proposition 3.10. P can be matched into T iff X_k is non-empty for some matching function F .

Proof sketch. \Rightarrow If there is a matching of P into T , then there is at least one matching function F for which X_k is nonempty:

The alternating run algorithm described in Section 3 does not find all possible matchings of P into T . For instance, in our running example (see Figure 1) the matching 4739 is not found by the algorithm since it involves elements that are not representative. Nevertheless, in case a matching of P into T described by a function $\varphi : [k] \rightarrow [n]$ exists, the algorithm detects another matching if it does not find φ itself and thus terminates successfully. We proceed in the following way: We first define a matching function F and then show that the corresponding X_1, \dots, X_k are non-empty.

In order to describe F , it is enough to determine the first element of every $F(i)$, $i \in [\text{run}(P)]$. Clearly, the first element of $F(1)$ is $T(1)$, i.e. the first element in T



We start with X_0 .

The only representative element (which is also the only valley) in $F(r(1)) = F(2)$ is 2, therefore $N_{1,(0,0,0)} = \{2\}$.

There are 3 representative elements larger than 2 in $F(r(2)) = F(1)$: 8, 4 and 3. Since 2 is not the largest element in its run in P_{ex} , condition (C2) implies that 3 is ruled out. Thus $N_{2,(0,2,0)} = \{8, 4\}$. Rule (R1) yields $X_2 = X'_2$. Rule (R2) is not applicable.

We have $N_{3,(8,2,0)} = \{11\}$ and $N_{3,(4,2,0)} = \{7, 6\}$ implying $X'_3 = \{(11, 2, 0), (7, 2, 0), (6, 2, 0)\}$. Rule (R1) discards $(11, 2, 0)$ in favor of $(7, 2, 0)$. Finally, Rule (R2) is applicable here and discards $(7, 2, 0)$.

The only representative element larger than 6 in $F(r(4)) = F(3)$ is the element 9. The matching of $P_{ex} = 2314$ into T_{ex} found by the algorithm is thus 4629.

Figure 5: The construction of X_0, \dots, X_4 for our running example (T_{ex}, P_{ex}) .

– cf. (P3). When determining $F(i)$, let j be the first element in i -th run in P . If the i -th run is a run up (down), the first element of $F(i)$ is given by $\min_T\{l : l \preceq_T \varphi(j) \text{ and } l \text{ is an element following a peak (valley)}\}$. This construction guarantees that the properties (P1)-(P4) are fulfilled. Constructing F in the described way for the matching 4739 yields the matching function presented in Figure 1.

Now we show that the described algorithm can indeed find a matching. This follows from the fact that the existence of a matching implies for every $\kappa \in [k]$ the existence of at least one element that fulfills conditions (C1) and (C2) and that is not removed by rule (R1) or (R2).

- (C1) The existence of $\varphi(\kappa)$ implies the existence of a representative element, as stated in the description of the algorithm, Part 2.
- (C2) Condition (C2) is fulfilled by $\varphi(\kappa)$ and its corresponding representative element.
- (R1) This rule always picks the smallest element in a vale. Since there is a matching involving $\varphi(\kappa)$, there is a matching containing the chosen element for this vale.
- (R2) Again, the chosen element is better than $\varphi(\kappa)$ since it is smaller or equal (the horizontal position does not matter here).

Thus the existence of φ guarantees that the data structure X_κ never is empty.

\Leftarrow If there is a matching function F such that the corresponding X_k is non-empty, then a matching of P into T can be found:

With the help of X_1, \dots, X_k we shall construct a matching $\psi : [k] \rightarrow [n]$ of P into T .

Start by picking some element $\mathbf{x} \in X_k$ and setting $\psi(k) := x_{r(k)}$. Now suppose the matching has been determined for $\kappa \in [k]$ to $\psi(\kappa) := y_{r(\kappa)}$ for some $\mathbf{y} \in X_\kappa$. Then there must exist an element $\mathbf{y}' \in X_{\kappa-1}$ that has lead to the element $\mathbf{y} \in X_\kappa$, i.e. \mathbf{y} differs from \mathbf{y}' only at the $r(\kappa)$ -th entry and $y_{r(\kappa)}$ satisfies the conditions (C1) and (C2) for \mathbf{y}' . We define $\psi(\kappa-1) := y'_{r(\kappa-1)}$. This defines the function $\psi : [k] \rightarrow [n]$. We now prove that ψ is a matching. First observe that ψ is a monotonically increasing map because of (C1). Second the sequence $\psi(P)$ is a subsequence of T . This is because of (C1) (correct order of elements in P within a run), (C2) and the concept of matching functions (correct order of elements in P in different runs). The rules (R1) and (R2) are only required for improving the runtime. \square

3.2. Runtime

Remark 3.11. In order to simplify the analysis of the algorithm we assume that if P ends with a run up (down) then T ends with a run down (up). This can be achieved by adding an element 0 ($n+1$) at the end of T . There is a matching of P into T iff P can be matched into the modified T . Furthermore, this element will never be chosen by the algorithm and therefore does not influence it. The only difference is that $\text{run}(T)$ increases by 1. This, however, does not show in the \mathcal{O} notation of the algorithm's runtime.

We first want to bound the size of X_κ for every $\kappa \in [0, k]$ and for a given matching function F . For the following definition recall the definition of vales (Definition 3.7).

Definition 3.12. For $i \in [\text{run}(P)]$ and $\kappa \in [k]$ let v_κ^i denote the number of distinct vales the elements in $\{x_i \mid \mathbf{x} \in X'_\kappa\}$ are contained in.

Lemma 3.13. Let F be a matching function, $i \in [\text{run}(P)]$ and $\kappa \in [k]$. If κ is not the largest element in its run in P then

$$v_\kappa^i \leq \frac{\text{run}(F(i))}{2}.$$

Proof. This is shown by induction over κ . Since $X_0 = \{(0, \dots, 0)\}$ the statement clearly holds for v_1^i with $i \neq r(1)$. For $v_1^{r(1)}$ six different cases have to be considered. Since this step of the proof is the same for all $v_\kappa^{r(\kappa)}$, we will show it for any $\kappa \in [k]$. These six cases are depicted in Figure 6. Continuous lines represent runs in $F(r(\kappa))$ that may contain elements in $\{x_{r(\kappa)} \mid \mathbf{x} \in X'_\kappa\}$. Dashed lines represent runs consisting of elements that are not considered because of condition (C2). Recall that (C2) stated that if κ was not the largest element in its run in P , only elements having larger elements to their right (left) were allowed to be considered for κ in a run up (down). In all cases represented in Figure 6 the number of considered vales is bounded by $\frac{\text{run}(F(i))}{2}$.

It remains to show the induction step. We already know that this step holds for $i = r(\kappa)$ (see Figure 6). For all other $i \in [\text{run}(P)]$ we only have to observe that $\{x_i \mid \mathbf{x} \in X'_\kappa\}$ is a subset of $\{x_i \mid \mathbf{x} \in X'_{\kappa-1}\}$. This is because the rules (R1) and (R2) remove elements

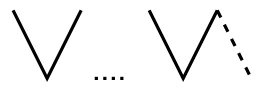
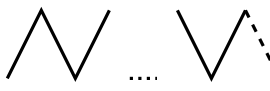
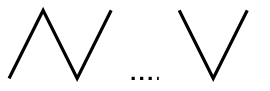

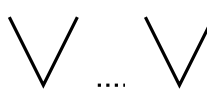

	Possible if κ is in the first run in P	Possible for arbitrary κ	Possible if κ is in the last run in P
κ is in a run up	 At most $\frac{\text{run}(F(r(\kappa)))-1}{2}$ vales are considered.	 At most $\frac{\text{run}(F(r(\kappa)))}{2}$ vales are considered.	 Not necessary to consider because of Remark 3.11.
κ is in a run down	 At most $\frac{\text{run}(F(r(\kappa)))-1}{2}$ vales are considered.	 At most $\frac{\text{run}(F(r(\kappa)))}{2}$ vales are considered.	 Not necessary to consider because of Remark 3.11.

Figure 6: All six different cases that have to be considered when counting the number of vales in $F(r(\kappa))$ for some $\kappa \in [k]$.

from $X'_{\kappa-1}$ and in the step from $X_{\kappa-1}$ to X'_κ only the $r(\kappa)$ -th component of elements in $X_{\kappa-1}$ is modified. \square

Lemma 3.14. *For any given matching function F and every $\kappa \in [k]$*

$$|X_\kappa| \leq \prod_{i=1}^{\text{run}(P)} \frac{\text{run}(F(i))}{2}.$$

Proof. We prove this statement by induction over $\kappa \in [0, k]$. First, it is true for $\kappa = 0$ since $|X_0| = 1$. Now, how many elements may be added in a step from $X_{\kappa-1}$ to X_κ ? We have to consider two cases:

1. κ is the largest element in its run in P :

In this case we apply the Rule (R2) as described in Section 3. As stated in Rule (R2) it is enough to consider a single representative element in $F(r(\kappa))$ for each element in $X_{\kappa-1}$. Thus every element in $X_{\kappa-1}$ leads to at most one element in X_κ and we have by the induction hypothesis:

$$|X_\kappa| \leq |X_{\kappa-1}| \leq \prod_{i=1}^{\text{run}(P)} \frac{\text{run}(F(i))}{2}.$$

2. κ is not the largest element in its run in P :

In this case it is Rule (R1) that guarantees the size bound for X_κ . From the

description of (R1) it follows that

$$|X_\kappa| = |\{[\mathbf{x}]_\sim \mid \mathbf{x} \in X'_\kappa\}|,$$

since a minimal element is determined for every equivalence class $[\mathbf{x}]_\sim$. Irrespective of how many elements are contained in X'_κ , we can give an upper bound for the total number of equivalence classes $[\mathbf{x}]_\sim$. By Definition 3.7 and 3.12 we know that

$$|\{[\mathbf{x}]_\sim \mid \mathbf{x} \in X'_\kappa\}| \leq \prod_{i=1}^{\text{run}(P)} v_\kappa^i.$$

Then Lemma 3.13 implies that

$$|X_\kappa| = |\{[\mathbf{x}]_\sim \mid \mathbf{x} \in X'_\kappa\}| \leq \prod_{i=1}^{\text{run}(P)} \frac{\text{run}(F(i))}{2}.$$

We have thus proven the desired upper bound for $|X_\kappa|$ for every $\kappa \in [k]$. \square

Proposition 3.15. *The runtime of the alternating run algorithm is $\mathcal{O}^*(1.784^{\text{run}(T)})$ and thus also $\mathcal{O}^*(1.784^n)$.*

Proof. We know from Lemma 3.14 that

$$|X_\kappa| \leq \prod_{i=1}^{\text{run}(P)} \frac{\text{run}(F(i))}{2} \tag{1}$$

for every $\kappa \in [k]$. Observe that

$$\sum_{i=1}^{\text{run}(P)} \text{run}(F(i)) = \text{run}(T) + \text{run}(P) - 1, \tag{2}$$

since two subsequent $F(i)$'s have one run in common (see Figure 2). The inequality of geometric and arithmetic means implies that this product is maximal if all $\text{run}(F(i))$ are equal, i.e. for every $i \in \text{run}(P)$

$$\text{run}(F(i)) = \frac{\text{run}(T) + \text{run}(P) - 1}{\text{run}(P)}.$$

Therefore X_κ has at most $\left(\frac{\text{run}(T) + \text{run}(P) - 1}{2\text{run}(P)}\right)^{\text{run}(P)}$ elements for every $\kappa \in [k]$. To find an upper bound for this function requires extensive calculations which we detail in Appendix A. It follows that $|X_\kappa| \leq 1.2611^{\text{run}(T)}$.

Given an element in $\mathbf{x} \in X_{\kappa-1}$ computing $N_{\kappa, \mathbf{x}}$ clearly requires only polynomial time. In total, computing X'_κ takes $\mathcal{O}^*(X_{\kappa-1}) = \mathcal{O}^*(1.2611^{\text{run}(T)})$ time. The size of X'_κ is then at most $|X_{\kappa-1}| \cdot \text{run}(F(r(\kappa)))$, since the conditions (C1) and (C2) do not hold for more

than $\text{run}(F(r(\kappa)))$ many elements per element in $X_{\kappa-1}$. Rule (R1) can be executed by sorting X'_κ and hence requires $\mathcal{O}^*(|X'_\kappa| \cdot \log(|X'_\kappa|)) = \mathcal{O}^*(1.2611^{\text{run}(T)})$ time. The same holds for Rule (R2).

By Lemma 3.5 there are at most $\sqrt{2}^{\text{run}(T)}$ matching functions F we have to consider. For each of these, X_0, \dots, X_k has to be computed. Thus the runtime of the alternating run algorithm is in $\mathcal{O}^*(\sqrt{2}^{\text{run}(T)} \cdot 1.2611^{\text{run}(T)}) = \mathcal{O}^*(1.784^{\text{run}(T)})$. Moreover it always holds that $\text{run}(T) \leq n$ and thus the runtime of the alternating run algorithm is also $\mathcal{O}^*(1.784^n)$ \square

3.3. The parameter $\text{run}(P)$

We present two XP results for PPM with respect to the parameter $\text{run}(P)$. The first result follows from the alternating algorithm when analyzing its runtime with respect to $\text{run}(P)$. The second result follows from an algorithm in [1] and a lemma shown in this paper.

Proposition 3.16. *The runtime of the alternating run algorithm is*

$$\mathcal{O}^* \left(\left(\frac{n^2}{2\text{run}(P)} \right)^{\text{run}(P)} \right).$$

Proof. In the proof of Proposition 3.15 we have seen that X_κ has at most $(\text{run}(T) + \text{run}(P) - 1)/(2\text{run}(P))^{\text{run}(P)}$ elements. We can of course assume that $\text{run}(P) < n$, since $\text{run}(P) \geq n$ would imply that P has more elements than T and then there clearly is no matching of P into T . At the same time $\text{run}(T) < n$ always holds. Thus $|X_\kappa| < (2n/(2\text{run}(P)))^{\text{run}(P)}$ for all $\kappa \in [k]$. Again, as in the proof of Proposition 3.15 constructing X_κ from $X_{\kappa-1}$ can be done in time $\mathcal{O}^*(|X_{\kappa-1}|)$, i.e. in time $\mathcal{O}^*((2n/(2\text{run}(P)))^{\text{run}(P)})$ and has to be done $|P| \leq n$ times.

Moreover we saw in Lemma 3.5 that the number of matching functions F is at most

$$\binom{\lceil \text{run}(T)/2 \rceil}{\text{run}(P) - 1} < (\lceil \text{run}(T)/2 \rceil)^{\text{run}(P)-1} < \left(\frac{n}{2}\right)^{\text{run}(P)}.$$

For each of these F 's, X_0, \dots, X_k has to be computed. Thus the runtime of the alternating run algorithm is $\mathcal{O}^*((n/2)^{\text{run}(P)} \cdot (n/(\text{run}(P)))^{\text{run}(P)})$ proving the desired XP runtime bound. \square

Before we can state the theorem proven in [1] which we will build upon, we need the following definitions.

Definition 3.17. *Given a pattern π of length m , the graph $G_\pi = (V, E)$ is defined as follows: The vertices $V := [m]$ represent positions in π . There are edges between adjacent positions, i.e. $E_1 := \{\{i, i+1\} \mid i \in [m-1]\}$. There are also edges between positions where the corresponding values have a difference of 1, i.e. $E_2 := \{\{i, j\} \mid \pi(i) - \pi(j) = 1\}$. The edge set is defined as $E := E_1 \cup E_2$.*

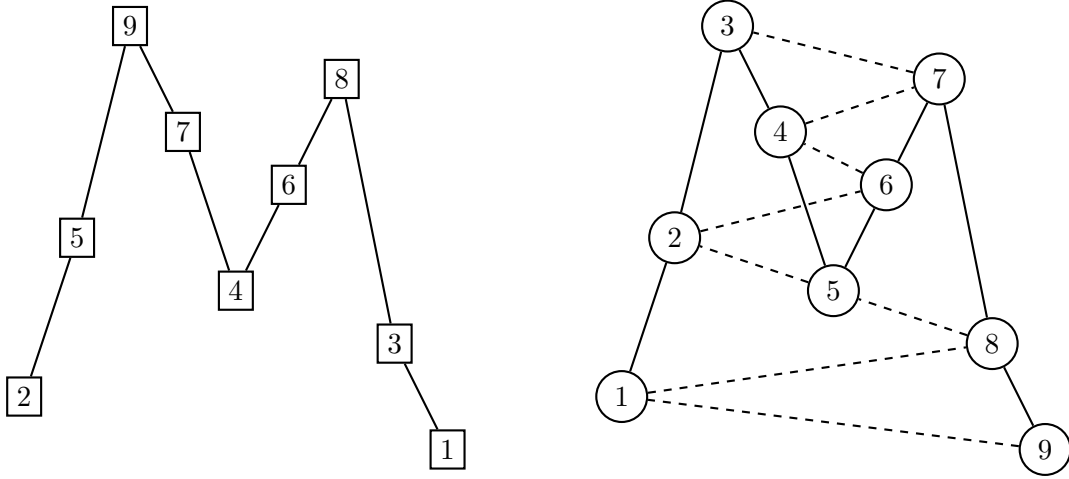


Figure 7: To the left a graphical representation of the permutation π introduced in Example 3.18, to the right the corresponding graph G_π

Example 3.18. Consider the permutation

$$\pi = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\ 2 & 5 & 9 & 7 & 4 & 6 & 8 & 3 & 1 \end{pmatrix}$$

written in two-line representation. A graphical representation of π can be found on the left-hand side of Figure 3.18. The corresponding graph G_π is represented on the right-hand side of the same figure. The solid lines correspond to the edges in E_1 and the dashed lines to the ones in E_2 . \dashv

Definition 3.19. Let $G = (V, E)$ be a simple graph, i.e. E consists of subsets of V of cardinality 2. A path decomposition of G is a sequence X_1, \dots, X_k of subsets of V such that

1. Every vertex appears in at least one X_i , $i \in [k]$.
2. Every edge is a subset of at least one X_i , $i \in [k]$.
3. Let three indices $1 \leq h < i < j \leq k$ be given. If $x \in X_h$ and $x \in X_j$ then $x \in X_i$.

The width of a path decomposition is defined as $\max\{|X_1|, \dots, |X_k|\} - 1$. The pathwidth of a graph G , written $\text{pw}(G)$, is the minimum width of any path decomposition.

Theorem 3.20 (Theorem 2.7 and Proposition 3.5 in [1]). PPM can be solved in time $\mathcal{O}(n^{1+\text{pw}(G_P)})$.

Lemma 3.21. $\text{pw}(G_\pi) \leq \text{run}(\pi)$ holds for all permutations π .

Proof. Given a permutation π of length m we will define a sequence X_1, \dots, X_m . We then show that this sequence is a path decomposition of $G_\pi = (V, E)$ with width at most $\text{run}(\pi)$. In this proof we use the variables i, j for positions in π and the variables u, v, w for values of π , i.e. $\pi(1), \pi(2)$, etc.

In order to define the sequence X_1, \dots, X_m of subsets of V , we shall extend alternating runs to maximal monotone subsequences. This means that we add the preceding valley to a run up and the preceding peak to a run down. For any $s \in [\text{run}(\pi)]$, R_s then denotes the set of elements in the s -th run in π together with the preceding valley or peak. Note that this implies that $|R_s \cap R_{s+1}| = 1$ for all $s \in [\text{run}(\pi) - 1]$.

We define $X'_1 := \{1\}$ and

$$X'_v := \{ \max(R_j \cap [v - 1]) \mid j \in [\text{run}(\pi)] \text{ and } R_j \cap [v - 1] \neq \emptyset \} \cup \{v\}$$

for every $v \in \{2, \dots, m\}$. Since X_v should contain positions (and not elements) in π we define

$$X_v := \{ \pi^{-1}(w) \mid w \in X'_v \}.$$

We now check that X_1, \dots, X_m indeed is a path decomposition.

1. The vertex i appears in $X_{\pi(i)}$.
 2. First we consider edges of the form $\{i, i + 1\}$. We distinguish two cases.
 - 1. case: $\pi(i) < \pi(i + 1)$.
Then $\{i, i + 1\}$ is a subset of $X_{\pi(i+1)}$. Clearly, $i + 1 \in X_{\pi(i+1)}$. Since $\pi(i)$ and $\pi(i + 1)$ are adjacent in π there has to be a $s \in [\text{run}(\pi)]$ such $\{\pi(i), \pi(i + 1)\} \subseteq R_s$. It then holds that $\max(R_s \cap [\pi(i + 1) - 1]) = \pi(i)$ since $\pi(i) \in R_s \cap [\pi(i + 1) - 1]$ and $\pi(i)$ is the largest element in R_s smaller than $\pi(i + 1)$. Consequently $i \in X_{\pi(i+1)}$.
 - 2. case: $\pi(i + 1) < \pi(i)$.
Analogously one sees that $\{i, i + 1\}$ is a subset of $X_{\pi(i)}$.
- Every edge $\{i, j\} \in E$ with $\pi(i) - \pi(j) = 1$ is a subset of $X_{\pi(i)}$: As before $i \in X_{\pi(i)}$. Let s be any element of $[\text{run}(\pi)]$ such that $j \in R_s$. Then $\max(R_s \cap [\pi(i) - 1]) = \max(R_s \cap [\pi(j)]) = \pi(j)$ and hence $j \in X_{\pi(i)}$.
3. Let $1 \leq u < v < w \leq m$ with $i \in X_u$ and $i \in X_w$. Let s be any element of $[\text{run}(\pi)]$ such that $i \in R_s$. Then either $\pi(i) \in R_s \cap [u - 1]$ or $\pi(i) = u$. In both cases is $\pi(i) \in R_s \cap [v]$. Furthermore, $\pi(i) = \max(R_s \cap [w]) = \max(R_s \cap [v])$. Hence $i \in X_v$.

The cardinality of each X_i is at most $\text{run}(\pi) + 1$ and hence $\text{pw}(G_\pi) \leq \text{run}(\pi)$. \square

Remark 3.22. This bound is tight since G_π for $\pi = 1\ 2\ 3 \dots m$ is a path and hence has pathwidth 1.

Example 3.23. Consider again π as defined in Example 3.18. The elements of the sets X'_1, \dots, X'_9 and those of X_1, \dots, X_9 as defined in the proof of Lemma 3.21 are given in Figure 3.23. It is easy to check that X_1, \dots, X_9 indeed is a path decomposition of

i	$\pi(i)$	X'_i	X_i
1	2	1	9
2	5	12	91
3	9	123	918
4	7	234	185
5	4	2345	1852
6	6	3456	8526
7	8	34567	85264
8	3	3 5678	8 2647
9	1	5 789	2 473

Figure 8: The sets X'_1, \dots, X'_9 and X_1, \dots, X_9 for the permutation $\pi = 259746831$

width $4 = \text{run}(\pi)$. Note that in the given table, columns of equal numbers do not contain any gaps. This fact corresponds to the third condition in the definition of path decompositions. \dashv

Corollary 3.24. PPM can be solved in time $\mathcal{O}(n^{1+\text{run}(P)})$.

4. W[1]-hardness for the parameter $\text{run}(P)$

The following hardness result shows that we cannot hope to substantially improve the XP results for PPM parameterized by $\text{run}(P)$ (Proposition 3.16 and Corollary 3.24). Indeed, an fpt algorithm with respect to $\text{run}(P)$ is only possible if $\text{FPT} = \text{W}[1]$.

Theorem 4.1. PPM is W[1]-hard with respect to the parameter $\text{run}(P)$.

Proof. The following proof is inspired by the proof given in [6] for the NP-completeness of PPM. We show W[1]-hardness by giving an fpt-reduction from the following problem to PPM:

CLIQUE

Instance: A graph $G = (V, E)$ and a positive integer k .
Parameter: k
Question: Is there a subset of vertices $S \subseteq V$ of size k such that S forms a clique, i.e. the induced subgraph $G[S]$ is complete?

Let (G, k) be a CLIQUE instance, where $V = \{v_1, v_2, \dots, v_l\}$ is the set of vertices and $E = \{e_1, e_2, \dots, e_m\}$ the set of edges. We are going to construct a PPM instance (P, T) . We shall first construct a pair (P', T') . P' and T' are both permutations on multisets, i.e. permutations in which elements may occur more than once. Applying Definition 2.1 to permutations on multisets means that in a matching repeated elements in the pattern have to be mapped to repeated elements in the text. Afterwards we deal

with the repeated elements in order to create a pattern and a permutation on ordinary sets and hereby obtain (P, T) .

Both the pattern and the text consist of a single substring coding vertices (\dot{P} resp. \dot{T}) and substrings coding edges (\bar{P}_i resp. \bar{T}_i for the i -th substring). These substrings are listed one after the other, with *guard elements* placed in between them to ensure that substrings have to be matched to corresponding substrings. For the moment, we will simply write brackets for the guard elements, indicating that a block of elements enclosed by a $[$ to the left and a $]$ to the right has to be matched into another block of elements between two such brackets. In addition, a *guard block* G_P is placed at the end of P . The construction of these guards shall be described later on.

We define the pattern to be

$$\begin{aligned} P' &:= [\dot{P}][\bar{P}_1][\bar{P}_2][\dots][\bar{P}_{k(k-1)/2}]G_P \\ &= [123\dots k][12][13][\dots][1k][23][\dots][2k][\dots][(k-1)k]G_P. \end{aligned}$$

\dot{P} corresponds to a list of (indices of) k vertices. The \bar{P}_i 's represent all possible edges between the k vertices (in lexicographic order).

For the text

$$T' := [\dot{T}][\bar{T}_1][\bar{T}_2][\dots][\bar{T}_m]G_T$$

we proceed similarly. \dot{T} is a list of the (indices of the) l vertices of G . The \bar{T}_i 's represent all edges in G (again in lexicographic order). Let us give an example:

Example. Let $l = 6$ and $k = 3$. Then the pattern permutation is given by

$$P' = [123][12][13][23]G_P.$$

Consider for instance the graph G with six vertices v_1, \dots, v_6 and edge-set

$$\{\{1, 2\}, \{1, 6\}, \{2, 3\}, \{2, 4\}, \{2, 5\}, \{3, 5\}, \{4, 5\}, \{4, 6\}\}.$$

represented in Figure 9 (we write $\{i, j\}$ instead of $\{v_i, v_j\}$).

Then the text permutation is given by:

$$T' = [123456][12][16][23][24][25][35][45][46]G_T.$$

—

Claim 1. *A clique of size k can be found in G if and only if there is a simultaneous matching of \dot{P} into \dot{T} and of every \bar{P}_i into some \bar{T}_j .*

Example (continuation). In our example $\{v_2, v_3, v_5\}$ is a clique of size three. Indeed, the pattern P' can be matched into T' as can be seen by matching the elements 1, 2 and 3 onto 2, 3 and 5 respectively. See again Figure 9 where the involved vertices respectively elements of the text permutation have been marked in gray. —

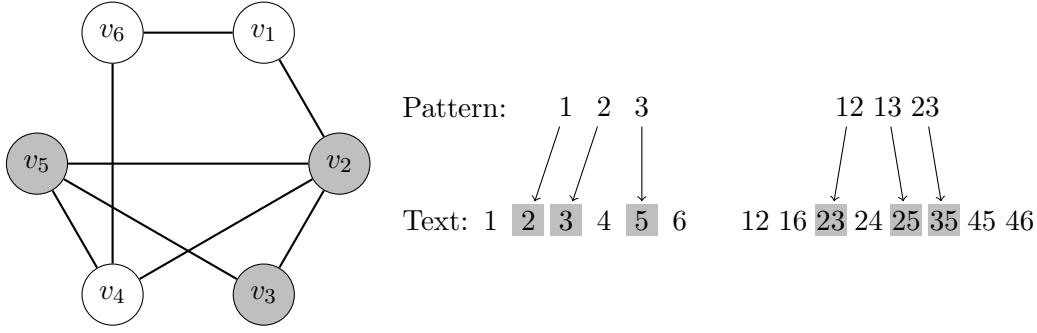


Figure 9: An example for the reduction of an INDEPENDENT SET instance to a PPM instance.

Proof of Claim 1. A matching of \dot{P} into \dot{T} corresponds to a selection of k vertices amongst the l vertices of G . If it is possible to additionally match every one of the \bar{P} 's into a \bar{T} this means that all possible edges between the selected vertices appear in G . This is because T' only contains pairs of indices that correspond to edges appearing in the graph. The selected k vertices thus form a clique in G . Conversely, if for every possible matching of \dot{P} into \dot{T} defined by a monotone map $\varphi : [k] \rightarrow [l]$ some $\bar{P}_i = xy$ cannot be matched into T' , this means that $\{\varphi(x), \varphi(y)\}$ does not appear as an edge in G . Thus, for every selection of k vertices there will always be at least one pair of vertices that are not connected by an edge and therefore there is no clique of size k in G . \square

In order to get rid of repeated elements, we identify every variable with a real interval: 1 corresponds to the interval $[1, 1.9]$, 2 to $[2, 2.9]$ and so on until finally k corresponds to $[k, k + 0.9]$ (resp. l to $[l, l + 0.9]$). In \dot{P} and \dot{T} we shall therefore replace every element j by the pair of elements $(j + 0.9, j)$ (in this order). The occurrences of j in the \bar{P}_i 's (resp. \bar{T}_i 's) shall then successively be replaced by real numbers in the interval $[j, j + 0.9]$. For every j , these values are chosen one after the other (from left to right), always picking a real number that is larger than all the previously chosen ones in the interval $[j, j + 0.9]$.

Observe the following: The obtained sequence is not a permutation in the classical sense since it consists of real numbers. However, by replacing the smallest number by 1, the second smallest by 2 and so on, we obtain an ordinary permutation. This defines P and T (except for the guard elements).

Example (continuation). Getting rid of repetitions in the pattern of the above example could for instance be done in the following way:

$$P = [1.9 \ 1 \ 2.9 \ 2 \ 3.9 \ 3][1.1 \ 2.1][1.2 \ 3.1][2.2 \ 3.2]G_P$$

This permutation of real numbers is order-isomorphic to the following ordinary permutation:

$$P = [4\ 1\ 8\ 5\ 12\ 9][2\ 6][3\ 10][7\ 11]G_P.$$

—

Claim 2. *P can be matched into T iff P' can be matched into T' .*

Proof of Claim 2. Suppose that P' can be matched into T' . When matching P into T , we have to make sure that elements in P that were copies of some repeated element in P' may still be mapped to elements in T that were copies themselves in T' . Indeed this is possible since we have chosen the real numbers replacing repeated elements in increasing order. If i in P' was matched to j in T' , then the pair $(i+0.9, i)$ in P may be matched to the pair $(j+0.9, j)$ in T and the increasing sequence of elements in the interval $[i, i+0.9]$ may be matched into the increasing sequence of elements in the interval $[j, j+0.9]$.

Now suppose that P can be matched into T . In order to prove that this implies that P' can be matched into T' , we merely need to show that elements in P that were copies of some repeated element in P' have to be mapped to elements in T that were copies themselves in T' . Then returning to repeated elements clearly preserves the matching. Firstly, it is clear that a pair of consecutive elements $i+0.9$ and i in P has to be matched to some pair of consecutive elements $j+0.9$ and j in T , since j is the only element smaller than $j+0.9$ and appearing to its right. Thus intervals are matched to intervals. Secondly, an element x in P for which it holds that $i < x < i+0.9$ must be matched to an element y in T for which it holds that $j < y < j+0.9$. Thus copies of an element are still matched to copies of some other element.

Finally, replacing real numbers by integers does not change the permutations in any relevant way. \square

It remains to implement the guards in order to ensure that substrings are matched to corresponding substrings.

At the end of P we place a guard block G_P consisting of a decreasing sequence of elements that are larger than all those occurring in P and that is at least as long as the permutation T (considered with guard elements). In P we have so far used $2k + k(k-1) = k + k^2$ integers and T consists of at most $2 + 2l + 2l(l-1) = 2 + 2l^2$ integers since there are at most $l(l-1)/2$ edges occurring in G and every edge i, j is encoded by the four elements $[ij]$. Therefore G_P is the decreasing sequence of $2 + 2l^2$ values $2 + 2l^2 + k + k^2, \dots, 2 + k + k^2, 1 + k + k^2$. As guard elements we use $k(k-1) + 2$ integers larger than the largest element previously used. The largest guard element is thus equal to $4 + 2l^2 + 2k^2$ and shall be denoted by P_{\max} . Then we insert the guards in the designated positions (previously marked by \lceil and \rfloor) in the following order: $P_{\max}-1$ (instead of the first \lceil), P_{\max} (instead of the first \rfloor), $P_{\max}-3$ (instead of the second \lceil), $P_{\max}-2$ (instead of the second \rfloor), \dots , $P_{\max}-2i-1$ (instead of the $(i+1)$ -th \lceil), $P_{\max}-2i$ (instead of the $(i+1)$ -th \rfloor), \dots , and so on until we reach the last guard-position. The guard elements are inserted in this specific order to ensure that two neighboring guard elements \lceil and \rfloor in P have to be mapped to two neighboring

guard elements \lceil and \rceil in T . We proceed similarly in T . G_T is the decreasing sequence of $2 + 2l^2$ values larger than the largest element of T . The sequence $T_{\max} - 1, T_{\max}, T_{\max} - 3, T_{\max} - 2, \dots$ where T_{\max} has been chosen accordingly is inserted in the guard-positions.

Claim 3. *Implementing the guards in the described way guarantees that in a matching of P into T , \dot{P} is matched into \dot{T} and each \bar{P} is matched into a \bar{T} .*

Proof of Claim 3. The two decreasing subsequences G_P and G_T added at the end of P and T indeed serve as guards. When the last element of P is matched into T , there are two possibilities. The first possibility is to map the last element either onto one of the large elements in G_T or to one of the guard elements in T . Then all larger elements in P have to be mapped to larger elements in T which means that all guard elements (including the decreasing subsequence in G_P) in P have to be mapped to guard elements in T . A pair $(i, i+1)$ of guard elements in P must be mapped to a pair $(j, j+1)$ of guard elements in T since $(j+1)$ is the only element larger than j and standing to its right. Consecutive guards in the pattern must be mapped to consecutive guards in the text, this implies that the elements in between these pattern guards must also be mapped to the elements in between the corresponding text guards. This is what we wanted: substrings have to be mapped to corresponding substrings.

The other possibility is to map the last element of P to some element x that is not a guard element in T . But then the remaining elements of P that are larger than the last one - these necessarily are more than $2 + 2l^2$ since this is the length of G_P - have to be matched into less than $2 + 2l^2$ elements in T since there are at most $1 + 2l^2$ elements lying to the left of x . This is not possible. \square

This finally yields that (G, k) is a YES-instance of CLIQUE if and only if (P, T) is a YES-instance of PPM. It remains to show that this reduction can be done in fpt-time. When counting the alternating runs in P we may consider two parts: the main part M_P of P consisting of all the non-guard elements as well as the guards placed in between them and the guard block G_P consisting of the decreasing subsequence of length $2 + 2l^2$. The length of M_P is equal to $k + k^2 + 2 + k(k-1) = 2 + 2k^2$ and thus clearly $\text{run}(M_P) \leq 2 + 2k^2 = \mathcal{O}(k^2)$. Since G_P consists of a single run down, it follows that $\text{run}(P) = \text{run}(M_P) + 1 = \mathcal{O}(k^2)$. Moreover the length of T is bounded by a polynomial in the size of G since $|T| \leq 4(1 + l^2) = \mathcal{O}(l^2) = \mathcal{O}(|G|^2)$. \square

Remark 4.2. Since in the fpt-reduction given in the proof of Theorem 4.1 the length of the pattern can be bounded by a polynomial in the size of G , this is also a polynomial time reduction. Therefore the proof of Theorem 4.1 can also be seen as an alternative way of showing NP-completeness for PPM.

5. Future work

Theorem 3.1 shows fixed-parameter tractability of PPM with respect to $\text{run}(T)$. An immediate consequence is that any PPM instance can be reduced by polynomial time preprocessing to an equivalent instance – a kernel – of size depending solely on $\text{run}(T)$.

This raises the question whether even a polynomial-sized kernel exists. Another research direction is the study of further parameters such as permutation statistics listed in the Appendix A of [14]. The major open problem in this regard is whether PPM is fpt with respect to the length of P . Finally, our method of making use of alternating runs might lead to fast algorithms for other permutation based problems.

6. Acknowledgments

We would like to thank the anonymous reviewers of SWAT 2012 for their feedback and suggestions leading to numerous improvements of this paper.

References

- [1] Shlomo Ahal and Yuri Rabinovich. On complexity of the subpattern problem. *SIAM J. Discrete Math.*, 22(2):629–649, 2008.
- [2] Michael Albert, Robert Aldred, Mike Atkinson, and Derek Holton. Algorithms for pattern involvement in permutations. In Peter Eades and Tadao Takaoka, editors, *Algorithms and Computation*, volume 2223 of *Lecture Notes in Computer Science*, pages 355–367. Springer Berlin / Heidelberg, 2001.
- [3] Michael H. Albert, Robert E. L. Aldred, Mike D. Atkinson, Hans P. van Ditmarsch, B. D. Handley, Chris C. Handley, and Jaroslav Opatrny. Longest subsequences in permutations. *Australasian Journal of Combinatorics*, 28:225–238, 2003.
- [4] Désiré André. Étude sur les maxima, minima et séquences des permutations. *Ann. Sci. École Norm. Sup.*, 3(1):121–135, 1884.
- [5] Miklos Bona. *Combinatorics of permutations*. Discrete Mathematics and Its Applications. Chapman & Hall/CRC, 2004.
- [6] Prosenjit Bose, Jonathan F. Buss, and Anna Lubiw. Pattern matching for permutations. *Information Processing Letters*, 65(5):277 – 283, 1998.
- [7] Mathilde Bouvel and Dominique Rossin. The longest common pattern problem for two permutations. *Pure Mathematics and Applications*, 17(1-2):55–69, 2006.
- [8] Mathilde Bouvel, Dominique Rossin, and Stéphane Vialette. Longest common separable pattern among permutations. In Bin Ma and Kaizhong Zhang, editors, *CPM*, volume 4580 of *LNCS*, pages 316–327. Springer, 2007.
- [9] Maw-Shang Chang and Fu-Hsing Wang. Efficient algorithms for the maximum weight clique and maximum weight independent set problems on permutation graphs. *Information Processing Letters*, 43(6):293–295, 1992.
- [10] Rodney G. Downey and Michael R. Fellows. *Parameterized Complexity*. Springer, 1999.
- [11] Jörg Flum and Martin Grohe. *Parameterized complexity theory*. Springer Berlin / Heidelberg, 2006.
- [12] Sylvain Guillemot and Stéphane Vialette. Pattern matching for 321-avoiding permutations. In Yingfei Dong, Ding-Zhu Du, and Oscar Ibarra, editors, *Algorithms and Computation*, volume 5878 of *Lecture Notes in Computer Science*, pages 1064–1073. Springer Berlin / Heidelberg, 2009.
- [13] Louis Ibarra. Finding pattern matchings for permutations. *Information Processing Letters*, 61(6):293–295, 1997.

- [14] Sergey Kitaev. *Patterns in Permutations and Words*. Springer Berlin / Heidelberg, 2011.
- [15] Donald E. Knuth. *The Art of Computer Programming, Volume I: Fundamental Algorithms*. Addison-Wesley, 1968.
- [16] Howard Levene and Jacob Wolfowitz. The covariance matrix of runs up and down. *The Annals of Mathematical Statistics*, 15(1):58–69, 1944.
- [17] Erkki Mäkinen. On the longest upsequence problem for permutations. *International journal of computer mathematics*, 77(1):45–53, 2001.
- [18] Rolf Niedermeier. *Invitation to Fixed-Parameter Algorithms*. Oxford Lecture Series in Mathematics And Its Applications. Oxford University Press, 2006.
- [19] Sanjeev Saxena and V. Yugandhar. Parallel algorithms for separable permutations. *Discrete Applied Mathematics*, 146(3):343–364, 2005.
- [20] Craige Schensted. Longest increasing and decreasing subsequences. *Classic Papers in Combinatorics*, pages 299–311, 1987.
- [21] Rodica Simion and Frank W. Schmidt. Restricted permutations. *European Journal of Combinatorics*, 6:383–406, 1985.

A. Appendix: Calculations leading to size bounds for the data structure X_κ

This appendix details the calculations in the proof of Theorem 3.15. We know that $|X_\kappa| \leq \prod_{i=1}^{run(P)} \frac{run(F(i))}{2}$ from Lemma 3.14 and

$$\sum_{i=1}^{run(P)} run(F(i)) = run(T) + run(P) - 1 \quad (3)$$

from the proof of Theorem 3.15. In the following we write $r := run(P)$ and $t := run(T)$.

From the inequality of geometric and arithmetic means it follows that

$$\prod_{i=1}^r \frac{run(F(i))}{2} \leq \left(\frac{t + r - 1}{2 \cdot r} \right)^r$$

and equality in the above inequation only holds when $run(F(i))$ is equal for all $i \in [r]$. Thus, $\prod_{i=1}^r \frac{run(F(i))}{2}$ is maximal for

$$run(F(i)) = \frac{t + r - 1}{r} \text{ for all } i \in [r].$$

Now we want to determine the maximum of the function

$$g(r) = \left(\frac{t + r - 1}{2r} \right)^r$$

that is an upper bound for $|X_\kappa|$.

$$\begin{aligned} g'(r) &= \frac{1}{r} \left(2^{-r} \left(\frac{r + t - 1}{r} \right)^{r-1} \cdot \left((r + t - 1) \log \left(\frac{r + t - 1}{r} \right) - r \log(2) - t(1 + \log(2)) + 1 + \log(2) \right) \right) \stackrel{!}{=} 0 \\ &\implies (r + t - 1) \left(\log \left(\frac{r + t - 1}{r} \right) - \log(2) \right) - t + 1 = 0 \\ &\implies \log \left(\frac{r + t - 1}{2r} \right) = \frac{t - 1}{r + t - 1}. \end{aligned}$$

The solutions are:

$$\begin{aligned} r_1(t) &= (-1 + t) / (-1 + 2e^{1+W_0(-1/(2e))}) \\ r_2(t) &= (-1 + t) / (-1 + 2e^{1+W_{-1}(-1/(2e))}), \end{aligned}$$

where W_0 is the principal and W_{-1} is the lower branch of the Lambert function defined by the equation $x = W(x) \cdot e^{W(x)}$. It holds that

$$\begin{aligned} (-1 + t)/3.311071 &\leq r_1(t) \leq (-1 + t)/3.311070 \\ (-1 + t)/-0.62663 &\leq r_2(t) \leq (-1 + t)/-0.62664, \end{aligned}$$

The second solution $r_2(t)$ is negative and therefore of no interest to us.

$$\begin{aligned}
g(r_1) &\leq \left(\frac{t + (-1 + t)/3.311070 - 1}{2(-1 + t)/3.311071} \right)^{(-1+t)/3.311070} \\
&\leq 0.80 \cdot \left(2.155535^{\frac{1}{3.311070}} \right)^t \\
&\leq 0.80 \cdot (1.261071)^t.
\end{aligned}$$

It therefore holds that

$$|X_\kappa| \leq 1.2611^{run(T)}.$$

Combining this with the fact that there are at most $\sqrt{2}^{run(T)}$ matching functions (as shown in Lemma 3.5) leads to the runtime:

$$\mathcal{O}((1.2611 \cdot \sqrt{2})^{run(T)} \cdot \text{poly}(n)) \leq \mathcal{O}^*(1.784^{run(T)}).$$